Seminar 5

1 Funcții template

Atunci când avem nevoie de o oferi aceeași funcție pentru mai multe tipuri de date, polimorfismul nu este întotdeauna bun (implementarea de mai multe ori ale aceleași funcții pentru diverse tipuri de parameterii nu este o practică foarte bună pentru a menține cod). Soluția este data de template-uri (sabloane).

O functie template este o functie în care tipul de date este parametrizat, deci putem avea multiple variante ale aceleasi functii scriind o singura data cod. Sintaxa pentru declararea unei functii template este:

```
template <typename T_1, typename T_2, ..., typename T_n>
<tip - retur> <nume - functie> (<lista - parameterii>) {
    /* corp functie */
};
```

In loc de `typename` putem folosi si `class` pentru a enumera tipurile de date parametrizate in template-ul nostru.
Exemplu:

```
#include <iostream>

template <typename T> T add (T a, T, b) {
    return a + b;
}

int main () {
    const int i = 4;
    const int j = 5;

    std::cout << add<int>(i, j); // instantiere explicita
    std::cout << add(i, j);      // instantiere implicita, compilatorul
                                 // deduce tipul de date
}
```

Codul asociat cu un template este compilat mai intai din punct de vedere al sintaxei. Compilarea semnatica se face la instanțiere. Orice eroare care ar putea rezulta datorită tipurilor de date cu care este instanțiat un template apare la instanțiere (daca nu instanțiem niciodata un template compilatorul nu va indica posibile erori de semnatica pentru acel template).

Pentru o funcție template putem oferi o specializare pentru anumite tipuri de date. Scopul unei specializari este sa oferim un comportament special pentru instantiere template-ului cu un anumit tip de date.

```
// specializarea functie template add pentru intregi
template <> T add <int> (int a, int, b) {
    std::cout << "integer specialization ";
    return a + b;
}

int main () {
    const int i = 4;
    const int j = 5;

```

```
    std::cout << add(i, j); // integer specialization 9
}
```

In C++, o functie template nu e vazuta ca o functie propriu-zisa, transformarea template-ului intr-o functie are loc la instantiere, cand compilatorul genereaza automat of functie in care tipurile parameterizate primesc valorie (e.g. add ¡int¿). Instantierea functie template care este marcata ca functie si poate fi folosita pentru apeluri.

Putem avea in clase non template metode (statice sau nu) si functii prieten template:

```
#include <iostream>
using namespace std;

class Point {
    int x, y;
public:
    Point (int a = 0, int b = 0) : x(a), y(b) {}
    friend ostream& operator<< (ostream&, const Point&);

    template <typename T>
    void foo (const T&);

    template <typename T>
    static Point convert (const T&, const T&);
};

template <typename T>
void Point::foo(const T& val) {
    x *= val;
    y *= val;
}

template <typename T>
Point Point::convert(const T& a, const T& b) {
    Point p;

    p.x = a; p.y = b;

    return p;
}

ostream& operator<< (ostream& out, const Point& p) {
    out << "(" << p.x << ", " << p.y << ")";
    return out;
}

int main () {
    Point p (2, 10);
    p.foo(2.4f);
    cout << p << endl;
    p = Point::convert(1.3f, 5.678f);
    cout << p << endl;
    return 0;
}
```

2 Clase template

C++ suporta si clase template pentru situatiile in care clasele pe care le scriem pot fi avea ca proprietati mai multe tipuri de date (e.g. stiva, vector, pereche, lista, matrice, hashmap etc.). Sintaxa pentru declarearea unei clase template este urmatoarea:

```
    template <typename T_1, typename T_2, ..., typename T_n>
    class <nume - clasa> {
        /* definitie clasa */
    };
```

Exemplu:

```
template <typename T> class Point;

template <typename T>
istream& operator >> (istream&, Point<T>&);


template <typename T> class Point {
    T x, y;
public:
    Point(const T&, const T&);
    int cadran ();

    template <typename U>
    void f (const U&);

    static void staticMethod ();

    template <typename U>
    static void templateStaticMethod (const U&);

    template <typename U>
    friend ostream& operator<<(ostream&, const Point<U>&);

    friend istream& operator>><T>(istream&, Point<T>&);
};

template <typename T>
Point<T>::Point(const T& a, const T& b) : x(a), y(b) { }

template <typename T>
int Point<T>::cadran() {
    if (x > 0) {
        if (y > 0) {
            return 1;
        } else if (y < 0) {
            return 2;
        }
    } else if (x < 0) {
        if (y < 0) {
            return 3;
        } else if (y > 0) {
            return 4;
        }
    }
    // the point is in origin or on axis
    return 0;
}

template<typename T>
template<typename U>
void Point<T>::f (const U& a) {
    cout << a;
}

template <typename T>
void Point<T>::staticMethod() {
    cout << "This is a static method in a template class";
}

template <typename T>
template <typename U>
void Point<T>::templateStaticMethod(const U& a) {
    cout << "This is template static method in a template class " << a;
}

template <typename T>
```

```cpp
ostream& operator<<(ostream& out, const Point<T>& p) {
    out << "(" << p.x << ", " << p.y << ")";
    return out;
}

template <typename T>
istream& operator>>(istream& in, Point<T>& p) {
    in >> p.x >> p.y;
    return in;
}

int main () {
    Point<int> p(0, 0);
    cin >> p;
    cout << p << " " << p.cadran() << endl;
    return 0;
}
```

Datorită modului în care sunt compilate template-urile, o clasă template nu poate fi scrisă în header (.h) și sursă (.cpp). Pentru a putea defini separat clasa trebuie să fie instanțiat template-ul la sfârșitul fișierului cpp.

Ca in cazul functiilor template, codul pentru clasa template nu denota un tip de date. Cand are loc instantierea template-ului compilatorul declara un nou tip de date. Fiecare instantiere a unei clase template creaza un nou tip de date care nu este legat in niciun fel de celelalte instantieri ale template-ului.

```cpp
template class Point <int>;   // instantiere explicita
Point<int> p;                 // instantiere implicita
```

Ca in cazul functiilor template, putem avea specializari pentru o clasa template, in care putem avea metode și proprietăți noi (care nu exista in definitia originala a clasei template).

```cpp
template <>
class Point <float> {
    /* template specialization */
};
```

3 Valori default in template

Ca in cazul parameterilor unei functii putem sa furnizam valori default pentru tipurile de date parameterizate intr-un template.

```cpp
#include <iostream>
using namespace std;

template <typename T = int>
struct Pair {
public:
    T first, second;
};

template <class T>
ostream& operator<< (ostream& out, const Pair<T>& p) {
    out << "(" << p.first << " ," << p.second << ")";
    return out;
}


int main () {
    Pair<> p1 = {2, 4};
    // in mod normal suntem obligati sa spune cu ce tip da date instantiem
    // dar in acest caz putem omite si compilatorul va stii ca e vorba de int
```

```
21      Pair<float> p2 = {3.4f, 10.6f};
22
23      cout << p1 << " " << p2 << endl;
24      return 0;
25  }
```

Exerciții

1. Implementați o clasă template pentru listă liniară înlănțuită, cu următoarea interfață:
   - constructor cu parameterii și de copiere;
   - metodă pentru adăugarea de element nou;
   - metodă pentru căutarea unui element (rezultat boolean);
   - metodă pentru ștergerea unui element după valoare;
   - supraîncărcarea operatorului [] pentru obținerea elementului de pe poziția `i` ;
   - supraîncărcarea operatorilor `<<` și `>>` pentru citire si afișare;
   - destructor;
   - operator de atribuire;
2. Implementați o clasa template pentru stiva, cu urmatoarea interfata:
   - constructor cu parameterii și de copiere;
   - metodă pentru adăugarea de element nou;
   - metodă pentru intoarcerea elementului din varful stivei;
   - metodă pentru ștergerea unui element după valoare;
   - supraîncărcarea operatorilor `<<` și `>>` pentru citire si afișare;
   - destructor;
   - operator de atribuire;
   - metoda care elimina 1 sau mai multe elemente din stiva si le intoarce sub forma de vector.